2008

CONDENSER: A custom tool for capturing and summarizing network traffic for AVALANCHE and ISEAGE

Jason Charles Maughan
Iowa State University

CONDENSER: A custom tool for capturing and summarizing network traffic for AVALANCHE and ISEAGE

by

Jason Charles Maughan

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Co-Majors: Information Assurance; Computer Engineering

Program of Study Committee:
Doug W. Jacobson, Major Professor
Thomas Earl Daniels
Johnny S. Wong

Iowa State University

Ames, Iowa

2008

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The Cyber-world is becoming an environment that is increasingly plagued by botnets and other programs adept at creating distributed and flooding-based attacks. These forms of assault are currently some of the most feared types of attacks being faced by networks.

The Internet Scale Event and Attack Generation Environment (ISEAGE) at Iowa State University, has been established to help researchers study actions and events that happen on the Internet. ISEAGE uses a custom tool, AVALANCHE, to simulate distributed and flooding-based attacks. While AVALANCHE is able to launch large quantities of attack packets, its ability to react and handle the enormity of responses from targeted hosts comes from the functionality afforded by the CONDENSER.

The CONDENSER uses multi-threaded artificial intelligence methods for efficiently catching the return traffic generated by targeted hosts, storing the payloads of captured packets, calculating statistics about the collection, reducing the captured payloads into a single representative-packet, and facilitating communication with remote command/control clients. These capabilities allow researchers to gain important insights into defensive measures and devices needed to stop distributed and flooding-based attacks.

CHAPTER 1. INTRODUCTION

In our modern age, our everyday activities and even existence often requires us to interact with computers. The invention and acceptance of the Internet has helped to further advance this dependence upon technology. No longer are we required to visit our local bank in order to check our accounts or transfer money. In fact, some banks have chosen to for-go building an actual building, and instead have moved all signs of their business exclusively to the cyber world. Long ago is the day when you had to visit the book store in order to purchase the latest bestseller. Now, one can simply login to an online account and have it sent to their front door. Perhaps, even more amazing is the fact that one may diagnosis a medical ailment by simply visiting a website.

While these capabilities allow for a great deal of personal convenience and an increased ability to reach an ever growing market place, they also create an often overlooked danger. This revolution of communication and technology has provided a means through which malicious attacks can be propagated to a greater than ever set of targets and victims. Things such as Denial of Service (DoS) attacks and their modern sibling Distributed Denial of Service (DDoS) attacks are becoming more and more threatening each day. Using the tactics of shear volume and overwhelming mass, they are able to affect the world around us by crippling websites and extorting information. This escalation in danger is in large part due to a phenomenon known as botnets, which will later be discussed.

Unfortunately, because of their shear size and scale, Denial of Service attacks are difficult for researchers to effectively study and ultimately protect against. Iowa State University's Internet Scale Event and Attack Generation Environment (ISEAGE) test-bed relies upon the CONDENSOR and the other portions of a custom tool called AVALANCHE, to help overcome problems related to the magnitude of flooding-based assaults.

AVALANCHE has been designed to function as much like a black-box device as possible. This means that aside from the arguments provided upon startup, AVALANCHE and its components don't require any additional user interaction in order to function. The majority of malicious assaults are effective because the aggressor has learned to automate them extremely well. Therefore, any tool that is used to simulate them, must also offer the maximum amount of automation as possible. The CONDENSOR exhibits this important characteristic and is able to carry out its functionality using artificial intelligence processes. These processes allow the CONDENSOR to store the payload of incoming packets, gather statistics about the collection, interface with the AMPLIFIER and a Command/Control Communication client, condense the received payloads into a packet representative of the polled collection, and accomplish all of these tasks in a timely manner.

Using the capabilities that have been briefly described, the CONDENSOR performs the indispensable task of supplying the feedback needed to help insure that a Denial of Service or botnet attack is able to be simulated without the use of thousands of computers or virtual-machine hosts. This helps to reduce the complexity inherent in the study of large numbers of attacking hosts, and allows for a larger number of attack vectors to be explored.

This thesis is organized so that Chapter 2 will focus on the Background information needed to understand the importance of the CONDENSOR and its important role in the ISEAGE environment. Next, Chapter 3 will further discuss the design and implementation aspects of the CONDENSOR. Finally, Chapter 4 will discuss possible future work and will contain the conclusion.

CHAPTER 2. BACKGROUND

There are many reasons and ideas that have motivated the need for the CONDENSER. In particular, this section will describe two of the security topics that the CONDENSER is particularly capable of addressing. This section will also describe the two technologies that will rely on the CONDENSER and its abilities. These technologies will provide a more complete set of tools available to researchers and information security professionals.

2.1 Botnets

The definition of a botnet is a group of interconnected bots. A bot can be summed up by referring to it as a computer running software which allows it to function autonomously and automatically. Under this definition a whole host of arguably benign pieces of software are considered bots. Something as simple as the programs that Google employs to discover and track new web pages, could and often are considered bots. However, the more pertinent meaning of the word botnet is a group of compromised computers running malicious software under a command and control type infrastructure. [1]

Focusing on the later definition of the term, it can be understood that a computer has to be compromised before it joins the ranks of the botnet. This process is often accomplished through the exploitation of a security vulnerability in the system's operating system or in one of the applications running on the system. Past botnet infections have used the same vulnerabilities as many of the major virus and worm outbreaks. Viruses and worms such as MyDoom, Netsky, Code Red, and Blaster, exploited the same weakness that were later used to infect computers and turn them into compromised machines commonly referred to as 'Zombies'[2].

Following their infection, zombies often make contact with a bot-controller. The bot-controller is the point from which commands, updates, and various other payloads are distributed to the rank-and-file bots. Once the newly converted bot has linked up with the bot-controller, it becomes capable of attacking a target. Interestingly enough, the infection process often includes the bot performing critical security updates to the infected host. This is an effort to keep other bots from taking over the computer, and thereby reducing the available bandwidth, processing power, and availability of the zombie.

After amassing large amounts of bots, an attacker can do several things. For instance, the attacker can use their botnet to infect more computers and thereby increase the size of their botnet even more. The bot-master, the attacker, may also choose to use their army of zombies, to begin attacking selected victims. While this process can be considered more of a game and ego trip to most bot-masters, it can also start to take on a more nefarious meaning.

Utilizing the amassed power of thousands of bots, bot-masters have been able to cross the threshold from game-oriented goals to more lucrative objectives. Using their bots to do things such as commit mass identity theft, manipulate Google's Adsense and other advertisement mechanisms, sniff network traffic, gather secret data using keyloggers, and send millions of spam emails, bot-masters have created a new playing field on the black market [3]. One additional means of revenue turns bot-masters into cyber-assassins, the Distributed Denial-of-Service Attack. Their potent capability can be hired to remove the presence of a competing website or any internet accessible network service.

It is the danger from Distributed Denial-of-Service attacks and the other malicious uses of these organized armies of computers that have driven researchers to confront botnets. Without continued and more extensive research, the bot-masters will continue to increase their capabilities and their tactics. In fact, there is already evidence of the use of botnets to get financial gain, morphing to become a tool of political and military attacks. For example, Estonia's networks were not that long ago attacked by a Russian botnet. The attack

precipitated in reaction to Estonia's decision to move a Soviet-era war memorial. Another similar attack involving a South Korean botnet and American-based servers happened during the 2002 Winter Olympics, in reaction to the Apolo Ohno fiasco [4]. Unfortunately, the frequency and severity of attacks like these will continue to escalate until either science can more fully address this problem or the war becomes lost.

2.2 Denial of Service Attacks

Perhaps some of the most overwhelming and most difficult assaults to defend against, Denial-of-Service attacks worry the most skilled of IT professionals. Yet, even more dreaded is the newest iteration of Denial-of-Service attacks, Distributed Denial-of-Service attacks. While, the Distributed Denial-of-Service (DDoS) attacks differ from Denial-of-Service (DoS) attacks, their overall characteristics are very similar and warrant more explanation.

A DoS attack is used to create a scenario where computer or network resources are unavailable to legitimate users. Often this scenario comes to fruition by saturating the target's system or network with a mass number of requests, so as to block out or severely hinder the ability of legitimate traffic to reach the besieged server or network [5]. While early DoS attacks relied upon exploitations inherent in the TCP/IP stack, newer DoS and DDoS attacks target the infrastructure and application layers of the network [6].

The infrastructure-layer is particularly susceptible to an attack method known as a SYN flood. A SYN flood uses an attack vector that focuses on taking advantage of the required steps used to establish a TCP connection. It is because of this reason that SYN flood type attacks are so incredibly difficult to thwart.

The three-step handshake starts by System 1 sending a SYN packet to a listening port on System 2. Once the listening port on System 2 receives the SYN packet, it moves to the SYN_RECV state and sends a SYN/ACK packet to the sending port on System 1. In order

for the connection to move to the ESTABLISHED state, System 1 must reply to System 2 with an ACK packet. However, a SYN flood attack finds the target host, System 2, stuck in the SYN_RECV state because the third step is never carried out. This attack is so effective because the connection in the SYN_RECV state requires the target system to allocate system resources to the setup. Since the target system never moves beyond the SYN_RECV state, the system resources can only be released after the connection-establishment timer expires. If the connection-establishment timer does not expire fast enough to keep up with the SYN attacks, all of the system's resources are consumed and the system becomes unresponsive. This assault is so devastating that even the largest of servers will succumb to the barrage generated by a relatively small attacker [6].

Application-layer attacks are geared to achieve the same end as infrastructure-layer attacks. Yet, application-layer attacks seek to exploit the legitimate uses of the target application, by overloading it with valid requests. In a nutshell, an application-layer strike finds the attacker locating an internet accessible service that allows a user to make "heavy" queries. A "heavy" query is one that requires a large amount of system processing to retrieve the results, but often require only a small amount of computation to make the actual request. An example of this would be a scenario that involves a phone book listing everyone on earth and a query for all the phone numbers that contained the number '5'. The query could be something as simple as "all PHONE_NUMBERS where contains 5 equals TRUE". Obviously, this query could easily be placed within a single packet and would require little parsing, but the results would require extremely expensive search and retrieval methods. To make matters worse, multiply the number of clients asking for this search from one to a thousand. In the end, this simple command would be an extremely devastating application-layer attack because of its limited computational requirements and the legitimacy of its request [6].

As this example illustrates, the application-layer attack vector carries with it much less overhead for the client-side, but delivers the same, if not greater, destruction to the server-side as infrastructure-layer assaults. For this reason, it can be even more devastating than the SYN-flood attack. Ultimately, if there is no reliable way to detect the attack and no way to survive the attack, then it can legitimately be called a "Perfect" weapon.

These two types of DoS attacks, application-layer and infrastructure-layer, have created havoc with their victims in the past. However, their potency has only become stronger with the increased use of DDoS attacks. DDoS attacks differ from DoS attacks because the actual attack packets originate from multiple sources and typically involve larger amounts of malicious traffic. This means that the attack source is incredibly difficult to track down, problematical to shutdown, and far richer in available resources. In a physical world example, one can imagine a DoS attack being similar to a supersonic jet fighter assaulting a stronghold. Imagining further, a DDoS attack would then be similar to possibly multiple squadrons attacking the same fortress from all different directions, using all different types of planes, and originating from different countries. Sure, the potential result is the same, a bombed out building. Yet, any defender would admit that they'd rather guard against a single foe no matter how menacing, as opposed to facing multiple enemies equally as dangerous and coming from multiple directions. In the second example, the odds of surviving are obviously drastically reduced if not completely zero.

It is the growing popularity of DoS attacks and more particularly DDoS attacks that bring a real need for continued research into defensive mechanisms. While the basic battle tactics of DoS and DDoS attacks are understood, there is value in the continued study of their effectiveness against defensive measures. For this reason, test-beds, which simulate an attack environment, have become and will continue to be critical to this crucial research effort.

2.3 ISEAGE

The Internet-Scale Event and Attack Generation Environment (ISEAGE) is an internet test-bed with a revolutionary design. Unlike many of the other research tools and environments that exist, ISEAGE has gone to great lengths to create a realistic atmosphere in which ideas and designs can be analyzed. This realistic setting gives a better understanding of how things work in the "real world."

In the past, most research has been carried out using computer simulations or simple test-networks. While this does give some idea as to the effectiveness and viability of a design, it lacks the small details and conditions that exist in a real world scenario. Like most complicated and dynamic systems, the Internet is bound to contain anomalies and other unanticipated events that can affect timings, traffic routes, network loads, and other characteristics. Therefore, one might conclude that the best method for testing the capability of a proposed design would be to attach it to the real Internet. Unfortunately, this too has its own draw backs.

The Internet is a dynamic system, and it is difficult to control or anticipate the types of traffic or data that cross-over it. This in turn leads to difficulty for researchers to accurately measure and evaluate the effectiveness of a project or solution. In addition to these short comings, introduction of a new device or protocol directly into the Internet also presents some level of danger to the Internet as a whole. What if the device were to turn rogue and begin disrupting malicious traffic elsewhere on the network or Internet?

It is to answer these dilemmas and problems that ISEAGE was created. Using a "highly configurable environment that can model any aspect of the Internet," ISEAGE is able to exemplify the Internet and its nuances without inheriting some of the headaches that exist in the real Internet [7]. Furthermore, the ISEAGE project has the added capability provided to it by custom research tools. "These tools create the virtual Internet and allow the researcher to introduce and monitor attacks [7]." Figure 1 is a schematic of ISEAGE and

shows the relationship between ISEAGE, the custom tools, and the systems/networks being tested [7]. The added research opportunities provided by ISEAGE will better help scientists and professionals to learn more about the measures need to offer greater protection and reliability to networks and devices.

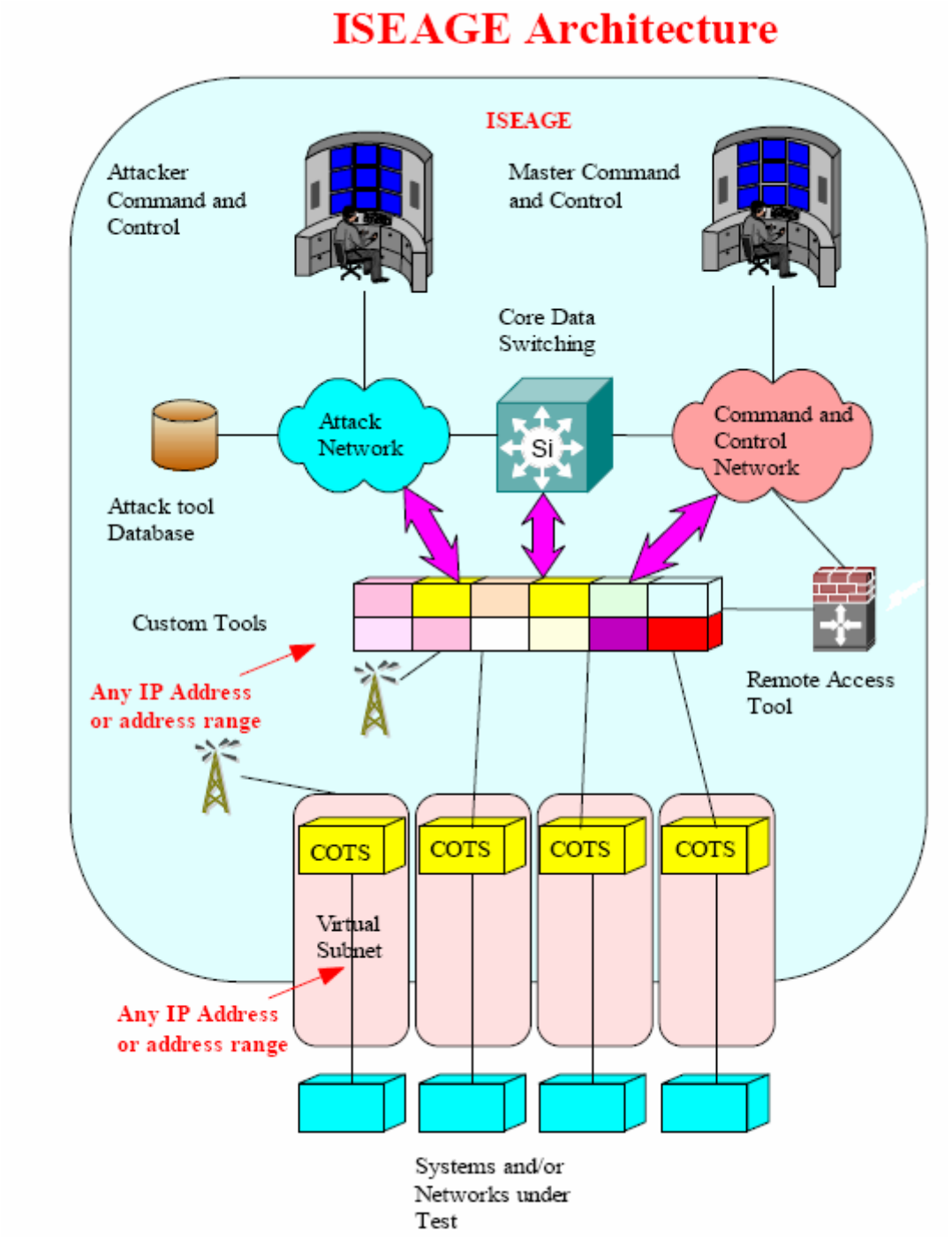

Figure 1. ISEAGE Architecture [7]

2.4 AVALANCHE

AVALANCHE is group of custom tools that enable researchers to study both distributed and flooding-based cyber-attacks. The increasing popularity of botnets, DDoS attacks, and other similar type assaults, necessitated the development of this toolset. This important addition to ISEAGE is comprised of a number of different components: the AMPLIFIER, the CONDENSER, and the Attack Plane. When functioning together, these tools will appear to be a "black box" to those researchers/programs that utilize their capabilities.

The AMPLIFIER tool is capable of magnifying a single attacker's packet into thousands of packets. This offers the first step in simulating attacks by botnets and other distributed or flooding-based assaults. The AMPLIFIER also introduces some potential for studying the difficulty and short comings of tracing back forged source IP addresses [7].

The CONDENSER is responsible for capturing all of the responses returned from the AMPLIFIER's targets. These responses are then sorted and condensed as needed, so that the one-to-many ratio between the attacker and its targets is maintained. The CONDENSER also gathers important statistical data that can be factored into the results being observed by the researchers.

The Attack Plane is the glue that joins AVALANCHE to ISEAGE. Since ISEAGE serves as a "miniature-sized" Internet, it is required to play by the same rules as the Internet. In particular, it must honor routing policies that govern the flow of IP traffic from Point A to Point B in the networking world. As was mentioned earlier, the AMPLIFIER relies upon the ability to forge source IP addresses when transmitting traffic; however, if the responses to those attack packets are to be gathered by the CONDENSER, then a routing capability must be implemented that can return only those packets relevant to the CONDENSER. If this ability does not exist, then replies could easily be lost or incorrectly delivered because of their fictitious destination address. The Attack Plane is designed to mark those packets that

pertain to AVALANCHE and thereby to retrieve the responses needed to allow AVALANCHE to work. This tracking capability is gained from the use of 'Routing by Injection.' In short, instead of relying on the addresses of the packets, the Attack Plane tracks the points where they both enter and exit the network. Upon reentry to the network the Attack Plane transfers the packets back to the place of their origin. Figure 2 diagrams the route that data follow between the Attacker, AVALANCHE, ISEAGE, and the Target(s).

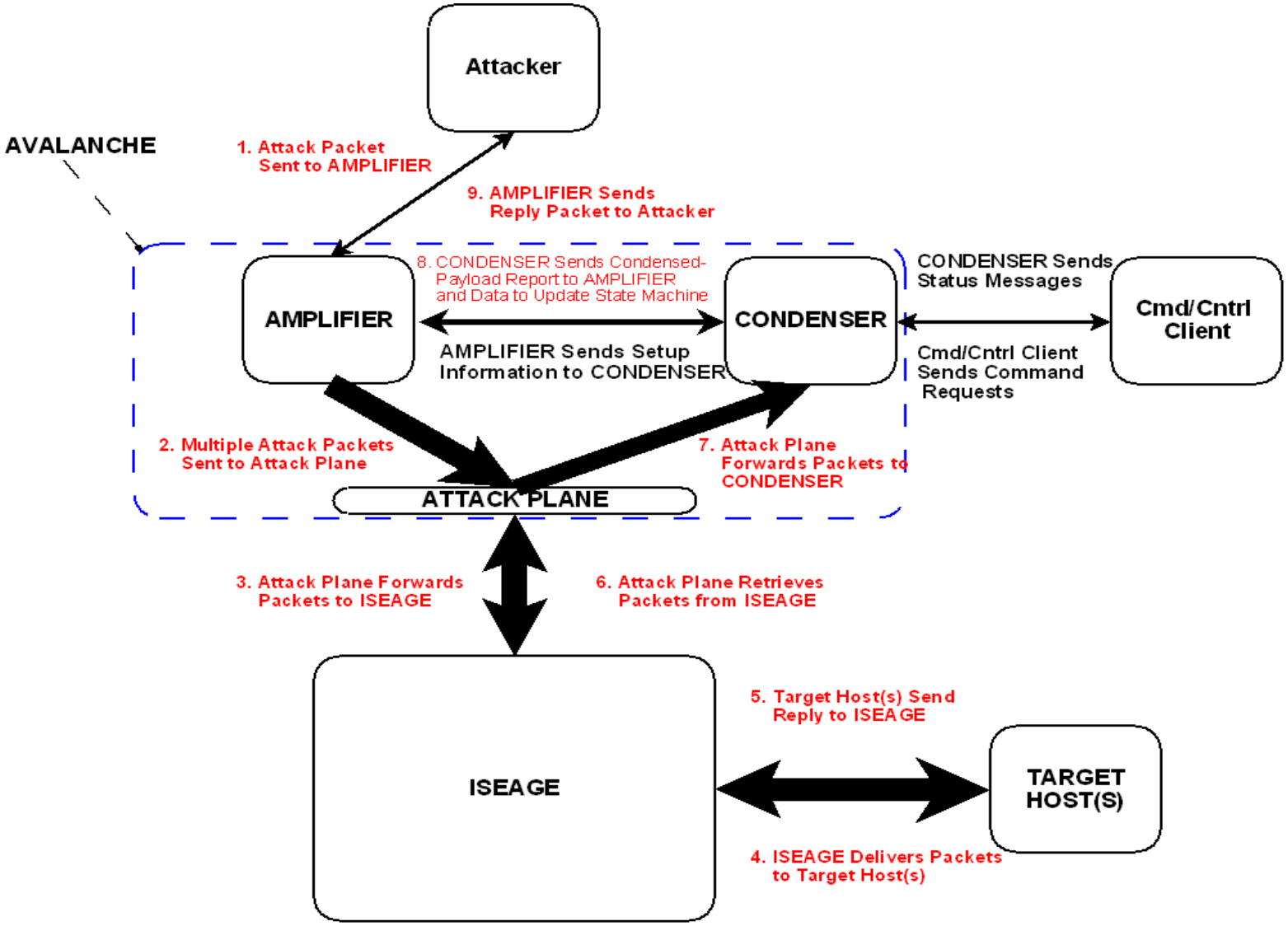

Figure 2. Information Flow through AVALANCHE

CHAPTER 3. DESIGN AND IMPLEMENTATION

This chapter describes the different portions of the CONDENSER design and their implementation. In particular, the chapter will describe how the various parts of the CONDENSER interface with each other, some of the problems that were overcome, and how each of the adopted solutions helps to meet the requirement introduced earlier. The chapter begins with an overview of the various components that make up the CONDENSER, and then it goes into an in-depth explanation of each of these components.

3.1 Overview

The CONDENSER program has been designed to meet an incredible challenge. The feat of processing a large number of inbound reply packets necessitated a more complex and robust design than a simple packet sniffer. Therefore, to meet this challenge it became quickly obvious that a single process would not be sufficient. The essential tasks' intricacy and speed requirements obliged the breaking down of the work load into seven main parts or components. Each of these parts functions interdependently with corresponding parts and works to create a synergy that allows for the accomplishment of the project's goals.

The seven parts that work together in order to make up the CONDENSER are the Master component, Capture component, Dissector component, Storage Handler component, Condenser component, Condenser/Amplifier Communication component, and the Command/Control Communication component. The Master component is responsible for the starting, the controlling, and the monitoring of all other components. The Capture component handles the gathering of incoming packets. The Dissector component takes the payloads that were gathered and sorts them according to each packet's contents. The Storage Handler component is responsible for the writing and reading of important captured payload data. The Condenser component evaluates all of the relevant payload data and determines how to best represent the acquired information. The Condenser/Amplifier Communication

component is responsible for passing keywords and information between the CONDENSER and the AMPLIFIER. Finally, the Command/Control Communication component allows for remote communication between the CONDENSOR and a remote user. Figure 3 is a diagram showing each of these components and the general flow of data through the CONDENSER.

While each of these individual parts is referred to as a component, it is actually implemented as a thread, and therefore will sometimes be referred to as a thread. By designing the CONDENSER to use multiple threads, it is able to benefit from many of the advantages common to concurrent programs.

Concurrent programming is a technique in computer science that allows for multiple instructions to be executed simultaneously. Imagine the following situation:

> "Several cars want to drive from point A to point B. They can compete for space on the same road and end up either following each other or competing for positions (and having accidents!). Or they could drive in parallel lanes, thus arriving at about the same time without getting in each other's way [8]."

This scenario was taken from the introduction to a concurrent programming book, and serves to illustrate the advantages of concurrent programming very well. It is obvious that concurrent programming can offer some advantages for helping to get all of the cars, in our case packets, to the finish line, or the AMPLIFIER. It is in an effort to acquire each of these advantages that the CONDENSER has been designed to use a multi-threaded architecture. However, these benefits also compel the addressing of a couple of different challenges unique to a multi-threaded environment, namely the critical section problem and the bounded buffer problem. Due to its presence in all of the components, the critical section problem and its solution are described here. The bounded buffer problem is left to be discussed along with the Capture component where it had to be addressed.

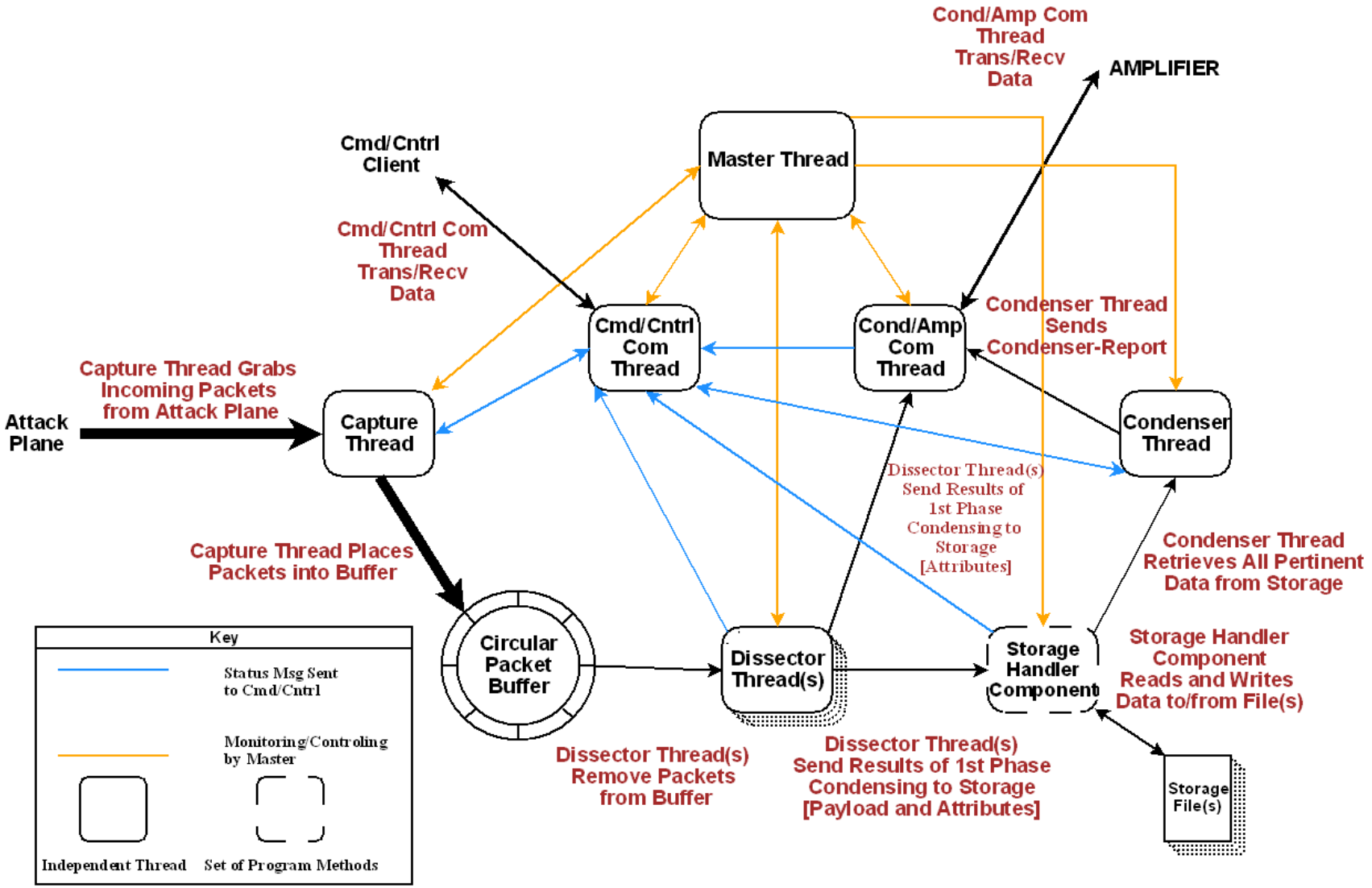

Figure 3. Information Flow through the CONDENSER

The critical section problem addresses the issue of variable value corruption common to concurrent programs that have not taken steps to safeguard the variable's values. Throughout the execution of a multi-threaded program, the individual threads may be interrupted by the processor, for a variety of reasons, and the processor may allow a different thread to begin executing. If variables and resources are not properly controlled, this "random" change in execution, can lead to one thread overwriting the work of another thread. This can be better understood through the use of a culinary example:

> Two blind and mute cooks are in the kitchen creating a salad and a cake. The first is responsible for the salad and the second the cake; however, each is only allowed to do anything when the judge tells them it is their turn. Once, he tells them to stop, they must set whatever they're working on down and wait for the word to start again. Furthermore, there is only one bowl to share between the two of them.
>
> One can imagine the outcome of this situation if the second cook has the bowl full of cake mix when the judge tells him to stop. The first cook resumes the creation of his salad by dumping each of the chopped vegetables into the bowl for tossing. Yet, before he can finish placing the "salad" on the plates, the judge tells him to stop and the second chef takes over. He's quick to grab his bowl of "cake mix" and dumps it into the pan so that it can go into the oven. In the end, neither chef is overly impressed with their results.

This disaster in the kitchen is a perfect example of the critical section problem. Without proper safeguards in place, it is extremely likely that a variable's value or a dessert can be ruined. Fortunately, this problem has a simple solution and the CONDENSER's components utilize this remedy.

Semaphores offer a great way to protect important data from being overwritten by other threads. It is important to note that the use of shared variables is not a bad thing, but much to the contrary, it is a great idea. Shared variables are a means through which multiple

threads can communicate. Just like in our culinary example, sharing a bowl is not a bad thing, as long each chef is done using it when the other takes over. Semaphores allow each thread to mark which variables are still in use, so that other threads do not corrupt them. In the chefs' kitchen this would be like sticking a lid on the bowl before your turn was over. If the other chef goes to use the bowl and the lids on it, then they leave the bowl alone and start prepping something else or simply wait. Once the bowl's lid is removed, the other chef knows that it is ok to mix their salad [8].

It is clear that the decision to use a multi-threaded approach brings with it a few additional challenges, but the remainder of this section and the actual functionality of the finished program will demonstrate how well the CONDENSER's tasks are addressed using concurrent methods.

3.1.1 Master Component

As was mentioned previously, early on in the design it became evident that the CONDENSOR would not be able to function as a single thread. Due to the fact that network traffic could not be counted on to come at a uniform rate and time, there would be instances when the capturing portion of the program had nothing to do. On the other hand, at other times there would be a lot of capturing to do, and there would also be communication required between the AMPLIFIER and the CONDENSER. Without the use of concurrent programming, this meant that either there would be packets dropped because they were not captured or that important communication would be sometimes severely delayed. Therefore, the need to transition into the concurrent programming world became increasingly obvious.

The use of multi-threaded programming brought its own synchronization challenges. The first obstacle that needed to be overcome involved the organization of the many different threads. Using an example from the physical world, one can imagine the making of a car. One man could certainly build the whole thing himself, but it would be a slow and

cumbersome process. So, instead there is a whole group of people charged with the construction of a car. One person is responsible for the frame, one handles the engine, another attaches the seats, another adds the tires, and so on. But, what happens if the worker in charge of the engine goes on break and forgets to tell anyone? If no one notices, the cars continue down the assembly line and are completed without ever receiving an engine. While this would save on gas mileage, it would not help to improve the company's bottom line. In order to avoid this debacle, companies employ bosses and managers. It is their job to coordinate the efforts of all the workers, so that the finished product works and functions as is designed.

In the case of the CONDENSER, the same reasoning stands. While it is not cars that are being produced, quality deliverables and standards still need to be achieved. How useful would the condensing of a bunch of packets be if, the condenser component was not aware of the fact that the Capture thread had gone on break? Therefore, it was extremely critical to include a thread specifically responsible for monitoring the status of all the other threads, and to react accordingly to any changes as they occurred. While not all synchronization of the components happens in the Master thread, it is responsible for the proper startup, shutdown, and 'emergency' shutdown of all other threads.

The Master thread is able to achieve this monitoring through the use of sentinel values. Upon startup of each of the processes, the Master thread clears the component's corresponding sentinel value. If at any point, one of the main components quits functioning, a sentinel value is changed and the Master thread knows to react. This reaction often requires the shutting down of the entire program. This is largely due to the strong interdependence between most of the components. However, this Master component shutdown also gives the other program elements a chance to gracefully conclude their work and to output any results that they're currently working on. In particular, this allows for all

of the packets that have been properly handled to complete as much of the process as possible.

For example, if the Condenser/Amplifier Communication thread is terminated, it means that the condensing results and/or other messages will not be able to be sent to the Amplifier. However, all of the Dissecting components that are currently working on sorting a packet will be able to save the packet's relevant payload to storage before exiting. In this way, researchers will still benefit from being able to review the information output by the Storage Handler component even though the CONDENSER had to be shutdown.

Using these sentinel values and controlling the start up of each of the other threads, the Master thread is able to add control and organization to the chaos that can often exist in multi-threaded programs.

3.1.2 Capture Component

The Capture component is responsible for the first step in the condensing process. This step involves the gathering or capturing of the packets that need to be condensed. The condenser system is currently designed to use an Ethernet network interface card, in order to receive all of the pertinent network traffic. However, in order to access the payload and other important information that is stored in a packet, demultiplexing must occur [9].

The demultiplexing procedure happens in much the same way that a letter is delivered to its recipient. First, the postal office refers to the portion of the letter that identifies which country it needs to go to. A Russian mail clerk really does not pay much mind to the portion of the address that lists 123 Main Street, but does concentrate on the country name of United States of America. Following its delivery to the United States, the letter's Zip Code is used to direct it even closer to its destination. Finally, the box number "123" is used to identify the specific mailbox that must be used to successfully deliver the letter. However, what

happens if the box “123” does not exist? Experience dictates that the outcome involves the letter being returned back to the sender in Russia with a note indicating its failed delivery.

This same scenario is played out when transmitting packets of information across a network. The popular OSI model is used to describe the different layers of protocols that are used to direct a packet from its source to its destination. Much like the letter sent from Russia, each layer attaches a segment of the overall address to the packet. As the packet gets closer to its destination, a different part of the address is used to further the delivery progress. Until finally, the application level address is used to deliver the packet to the waiting program. In the event that there is not a program waiting for the packet, the response process is similar to the physical letter. One of two things happens, a note is returned to the sender indicating the delivery failure, or the packet is simply discarded.

Since the CONDENSER needs to capture all incoming packets, it would seem that an application would have to be waiting at the destination address of every response packet sent to the CONDENSER. Since, the AMPLIFIER could possibly send thousands of packets with completely random and changing return addresses; it is difficult to imagine how to capture every one of those packets. In fact, it would be incredibly difficult to capture each packet using this method, but fortunately there exists another way of getting the data that is needed.

Using the *libpcap* API the Capture component is able to make copies of the packets as they move through the Data Link layer. Then it can sort and parse these copies into different categories dependant upon their characteristics. It is the digital equivalent of making a copy of all the mail that goes through the post-office, and then sorting it based on a set of its physical properties.

This sorting process takes place in two phases. The first phase is handled by the Capture component, and the second phase takes place in the Dissector component of the program. The second phase of sorting will be discussed a little later, but the first phase operates using the capabilities of the BSD Packet Filter (BPF). Since, the *libpcap* API allows

for a direct interface with the BPF, the Capture thread is able to load an optimized filter specific to its needs.

The reason this design decision makes particular sense is because the use of the BPF has one distinct advantage over a developer-created application-layer filter. The advantage is found in that the BPF works at the kernel level. This means that only those packets that pass the first phase of sorting are copied to the user space of the operating system. This helps to avoid many of the unneeded and expensive writes that would happen if the first phase of sorting did not take place at the kernel level, but instead took place in the Capture thread [10].

In order to configure the BPF, the filtering criteria must be described using the keywords understood by the BPF. So that the syntax can be better understood, the following would be an acceptable filter expression to describe the previously mentioned conditions: "`tcp and udp and not host 192.168.1.100 and not host 192.168.1.101 and not src host 192.168.1.102`"

In this example, the IP Address "192.168.1.102" is the address of the capturing device. The other addresses, "192.168.1.100" and "192.168.1.101" are the addresses used by the Condenser/Amplifier Communication Thread and the Command/Control Communication Thread. The expression states that all packets that are either TCP or UDP type packets, do not contain a host address of "192.168.1.100" nor "192.168.1.101", and do not have a source (SRC) address of "192.168.1.102", should be allow to pass through the Phase 1 sorting filter.

So that the user can construct and specify their own BPF by using the start-up option "-f", a list of those keywords is included in Appendix A. If a user specified filter expression is not supplied, the CONDENSER constructs its own expression. This default expression requires that all accepted packets are either UDP or TCP packets, that they are not coming from or destined to any of the non-capturing network interface cards installed on the system,

and that they have not originated from the card that is currently collecting all of the network traffic.

While this expression will do an effective job of eliminating those packets that should not be condensed, the filter needs to be associated with a user specified capture device. The capture device is one of the network interface cards (NICs) that have been installed in the computer. The user specifies which device by supplying the device name. This name can be found by issuing the following command at the system prompt:

```
condensor@Avalanche-Cond:~$ ifconfig
```

The command will return a list of all the currently installed network devices that the system recognizes. In Figure 4, the portion that is highlighted refers to the device name of one of the installed devices.

The CONDENSER requires the user to specify the capture device instead of simply relying on the default action of choosing the first device, so that different hardware configurations can be used. Another reason is because future work will allow the CONDENSER to receive incoming packets using multiple capturing devices and different application-specific BSD packet filters. Once the user has determined the device name, it can be supplied to the CONDENSER using the command argument "–i", or if not supplied on the command line the user will be prompted for input upon CONDENSER startup.

Once, the BPF and capturing device have been set up, the Capture component is able to copy packets off the wire and sort them. Once the sorting has been completed the Capture thread deposits each packet into a circular buffer. The circular buffer is controlled using semaphores, which helps to ensure that no packet is lost by overwriting or other forms of memory corruption. Unfortunately, this design also creates the issue of the bounded buffer problem.

```
condensor@Avalanche-Cond:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:19:5B:73:AC:1F
          inet addr:192.168.1.102  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::219:5bff:fe73:ac1f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:162 errors:0 dropped:0 overruns:0 frame:0
          TX packets:39 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:20604 (20.1 KB)  TX bytes:4713 (4.6 KB)
          Interrupt:20 Base address:0xa000

eth1      Link encap:Ethernet  HWaddr 00:30:48:73:B2:E6
          inet addr:192.168.1.100  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::230:48ff:fe73:b2e6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:162 errors:0 dropped:0 overruns:0 frame:0
          TX packets:150 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:110597 (108.0 KB)  TX bytes:16289 (15.9 KB)
          Base address:0xa800 Memory:f2000000-f2020000

eth2      Link encap:Ethernet  HWaddr 00:30:48:73:B2:E7
          inet addr:192.168.1.101  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::230:48ff:fe73:b2e7/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:57 errors:0 dropped:0 overruns:0 frame:0
          TX packets:102 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:9342 (9.1 KB)  TX bytes:14509 (14.1 KB)
          Base address:0xac00 Memory:f2020000-f2040000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

Figure 4. Ifconfig Output

The bounded buffer problem refers to the situation when a producer process and consumer process work at different rates. For instance, the producer process may deal with bursty data transmissions, and therefore may alternate from creating a large number of items to no items at all. The consumer thread on the other hand, works at a steady rate processing the data items as they become available. Moving back to the culinary world, this is similar to

a baker and their customer. It takes the baker 60 minutes to make a sheet of rolls, but only takes the customer a few minutes to eat one roll. If the customer can wait for the initial batch of rolls to come out of the oven, the baker can make enough rolls every hour to keep the customer full for the entire day. All the baker needs to do is continue to make rolls and place the freshly baked ones in their display case. The process continues when the customer has finished eating their roll, and goes to the case to select another roll.

In the baking example the bounded buffer is the display case. The baker is able to add several rolls to it at a time, but the customer can only consume them one at a time. Therefore, it is critical that the display case is large enough to store enough roles so that the customer does not go hungry, and the baker has enough room to put the new rolls. To complicate the requirements, space in the bakery is limited and therefore the display case should only be as big as it needs to be. In practical terms, if the buffer is not large enough, the Capturing thread could run out of buffer space to store incoming packets, which means that it will be forced to wait for the consuming process to handle some of the stored packets. This in turn will lead to the dropping of packets by the BPF device.

In order to avoid the dropping of packets, the default buffer size is set to 2048 packet slots. This number was chosen after observing different buffer sizes during testing and the speed at which they filled up. 2048 packet slots is big enough to handle the load required by the NIC currently being used, and does not impose too high of a memory requirement on the system, only about 3 MB. Furthermore, the buffer size can still be increased by the user if the Capture thread is producing too many packets too fast, or decreased, if extra memory is needed for other components or processes. This adjustment can be made at startup using the "–x" option. Once the packets are stored in the buffer, they are ready to be retrieved by the next component.

3.1.3 Dissector Component

The Dissector component is the next step in the condensing process. Since, it is also one of the most processor intensive steps; it has been designed to allow for multiple copies of the thread to independently process a received packet. The Master component has been set to initiate 10 Dissector threads upon start-up. This value was arrived at after several testing exercises, using less and more threads, showed that packets were more likely to be dropped with other values.

The actual processing of the Dissector component starts with the thread retrieving a packet from the circular buffer. This packet is then subjected to the second phase of sorting. Following the sorting, the packet is handled according to the designation that was assigned to it during the classification process.

Before the packet can be retrieved from the circular buffer each Dissector thread must wait to enter the critical section that protects the buffer. As was described earlier, semaphores serve as the synchronization points between the Dissector threads and the Capture component. Furthermore, the critical section also helps to ensure that only one of the Dissector threads retrieves a particular packet. Once the packet has been retrieved from the buffer, that slot is then freed for a new packet to be placed in it by the Capture thread.

The second phase of sorting requires the packet to be classified according to its properties. The first property that is examined pertains to the protocol type of the packet. This serves to identify a UDP or TCP packet, which are currently the only protocols that the CONDENSER is capable of handling.

As a result of the protocol identification stage, a TCP packet is separated into one of five categories based on the size of the payload and the values of its flags, and a UDP packet is moved into the UDP group. The next six sub-sections describe each of the subsequent categories into which TCP and UDP packets are moved and processed. In particular, the sub-sections elaborate on the data structures and methods used to relay the important

information describing each packet to the other pertinent CONDENSER components. Appendix B contains a graphic which illustrates the data structures used by the Dissector components.

3.1.3.1 First Step of TCP Handshake

To be sorted into this category the packet must be a TCP packet, and have only the SYN flag set. This type of packet is used when a TCP socket is attempting to establish a connection with a remote host. The response to this type of packet requires that the SYN and ACK flags both be set in the response packet. Since the CONDENSER is only responsible for the reception of packets, and not the sending, the response packet needs to be generated by the AMPLIFIER.

Having noted that the AMPLIFIER is responsible for creating and sending the response packet to the source address, the AMPLIFIER will require some data from the received packet. In particular, the AMPLIFIER will need to know the source and destination IP address, source and destination port numbers, the sequence number, the acknowledgement number, and the window size. All of this information is located in the IP and TCP headers, and will be retrieved from the received packet.

Once it is copied from the headers, the information is organized into a data structure and sent immediately to the Condenser/Amplifier Communication component. The data structure is identified by the keyword, “HANDSHAKE1.” This automatic forwarding of information to the Condenser/Amplifier Communication component is aimed at expediting the handshake process, and allowing the AMPLIFIER the capability of maintaining as many TCP connections as possible. Also, it is important to note that these types of packets only receive one level of condensing before being forwarded to the AMPLIFIER. The condensing comes as a result of the dropping of all of the other pieces of data in the header, which were not needed by the AMPLIFIER and therefore extraneous.

3.1.3.2 Second Step of TCP Handshake

This category's packets are similar to those found in the previous category. Nevertheless, they differ in a unique way. Their variation is found in the setting of the ACK bit. This means that this type of packet has both the SYN and the ACK bits set, but typically does not contain any data aside from the packet headers. In fact, it is the same type of packet that the AMPLIFIER transmits in response to the *First Step of TCP Handshake* type packet.

The data and information that is collected and sent to the AMPLIFIER is the same as those transmitted for *the First Step of TCP Handshake* type of packet. However, there is a slight difference in the data structure used to convey the information to the AMPLIFIER. The keyword, "HANDSHAKE2" instead of "HANDSHAKE1," is used to announce this category of packet.

This class of packet, much like the one previously described, receives only one level of condensing and is immediately sent to the AMPLIFIER via the Condenser/Amplifier Communication thread. While this process of condensing doesn't actually reduce the number of bytes needed to represent the received packet, it does convert it into a more manageable form for the AMPLIFIER by removing any extra data that is not needed. Therefore, it helps to reduce the overhead incurred by the AMPLIFIER when tracking updates for the individual connection states.

3.1.3.3 Reset or Finish

The packets that are sorted into this group have the RST flag, the FIN flag, or both flags set. Most of the time these types of packets only contain the information stored in their headers, but occasionally they will contain important payload data which can be further condensed and more importantly stored. The packets are typically transmitted when a connection has either been refused or is in the process of closing down. Consequently, they are incredibly helpful to the AMPLIFIER and need to be forwarded without delay.

This type of packet is important because the AMPLIFIER can update the state of the connection, and can quit using resources to transmit more packets to the target. Furthermore, this type of packet is also an excellent indicator that the attack payload was unsuccessful at penetrating the target destination.

In order to facilitate the packet's transmission to the AMPLIFIER, the Dissector thread gathers the needed data from the packet's payload and immediately sends it to the Condenser/Amplifier Communication thread. The source and destination IP addresses, and the source and destination port numbers are organized into a data structure with the keyword, "RESET."

3.1.3.4 Acknowledgement

The packets that only have the ACK flag set, and not the SYN flag, are placed into this group. They are then examined to determine if they are carrying a payload. This assessment is important because the TCP suite allows acknowledgment packets to also carry the next set of data to the destination host. This rule in the protocol allows for the piggy-backing of information and ultimately cuts down on the bandwidth needed to support the connection between the source and destination hosts.

If the packet contains any data, in addition to the protocol headers, the data is parsed out and converted into a string representation of the data's hex values. This means that a data sequence "HELLO" would be converted into the string "48454C4C4F" because "HELLO" has a hex code of "48454C4C4F." Once the payload has been converted, it is sent to the Storage Handler component along with the packet's source IP address, source port number, and summary of the TCP flags. In addition to this information, the Storage Handler component is sent the exact time in milliseconds when the packet was processed. This timestamp will help during the payload condensing process.

Although, the conversion of the payload section effectively doubles the amount of memory required to store the payload, it helps to ensure that none of the non-printable data bytes in the payload are lost. While special escape characters could have been used to accomplish this same task, the current solution allows for a much simpler implementation, which is easier to debug and maintain. Furthermore, by converting the payload to a string version of the hex values, the condensing algorithm is able to minimize the amount of memory required during the payload condensing process.

Finally, regardless if the packet contained a payload or not, the Condenser/Amplifier component is sent a data structure with the keyword, "ACKPACK." Inside the data structure is stored the source and destination IP addresses, the source and destination port numbers, the sequence number, the acknowledgement number, and the window size. Once the AMPLIFIER receives this data structure, it will be able to verify that a transmitted packet was received by the target host.

3.1.3.5 Payload Only

All of the remaining TCP packets that have not been classified into one of the other groups are sent to this category. Once they are here, the packet's payload is examined. If there is actual data besides that which is stored in the headers, the data undergoes the conversion process to create a string representation of the data's hex values. This string, a timestamp, the packet's source IP address, the packet's source port number, and a summary of the packet's TCP flags, are all sent to the Storage Handler component for logging to disk. If the payload is non-existent and all that is received is essentially a NO-OP packet, than the Dissector thread will simply skip this packet and continue to the next one without sending anything to the Storage Handler component. Regardless of the existence of the payload, the Condenser/Amplifier Communication component is sent a PAYPAC data structure. Inside this packet are the source and destination IP addresses, the source and destination port

numbers, the sequence number, and window sizes. This will allow the AMPLIFIER to acknowledge the receipt of the packet, thereby stopping the target host(s) from sending a repeated packet.

3.1.3.6 UDP Packet

All of the UDP type packets are sent to this category, where they are processed. Since UDP does not require the amount of overhead and setup that TCP does, there are not any flags of which the CONDENSER needs to be aware. However, the payload size is important. If there exists a payload, than the UDP packet is treated much like the *Payload Only* and *Acknowledgement* types of packets. This means that the payload is converted into a string representation. Then the packet's source IP address, the packet's source port number, timestamp, converted payload, and finally the string "UDPPAC", which is used in place of the TCP flags, is sent to the Storage Handler component.

Despite the existence of a payload (or nonexistence of a payload), the Condenser/Amplifier component receives a data structure with the keyword, "UDPPAC." Inside this data structure is the source and destination IP address of the received packet, as well as the source and destination port number of the packet. This information will ultimately allow the AMPLIFIER to update the state of the UDP connection.

3.1.4 Storage Handler Component

The Storage Handler component is the next logical step in the condensing process. Unlike the other Condenser components, the Storage Handler component is not itself a thread, but a set of methods that are called by either the Dissector threads or the Condenser component. These methods allow for the writing and reading of packet payloads and their characteristics, to and from disk. This is particularly helpful to ensure that researchers will have a record of all of the data carrying packets that were processed by the CONDENSER.

In order to guarantee effective reads and writes, the Storage Handler methods rely upon two things. First, once a write or read method is called, an ID number must be supplied. The ID number is used to reference a specific storage file and matches either the ID of the Dissector thread that is making the call, or it matches the query that the Condenser component is using to gather needed information. The CONDENSER relies upon a design that allows for multiple storage files, in order to minimize the likelihood of any bottle necks occurring in the condensing process. This minimization is realized by allowing each Dissector its own storage file, and by permitting the Condenser thread to access one file at a time. The second method, for assuring efficient reads and writes, is each read and write operation is located within a critical section. As in the other CONDENSER components, the Storage Handler relies upon semaphores to help to ensure that the Condenser component is always retrieving reliable data, and not interrupting a Dissector thread's writing operation.

Aside from efficient reading and writing, the Storage Handler has a number of other design features to facilitate the persistent storage of data carrying packets. For example, since it is possible that a storage file may be read more than once for condensing purposes, it is necessary for the program to keep track of what packets have already been read by the Condenser component. This facet is especially useful if the user decides to use the available command line option, "-A", which allows for appending to existing storage files instead of truncating and overwriting them.

This tracking feature was originally designed to use the timestamp values that were assigned to each stored packet. The retrieved time value was compared against a stored reference time. If the value was larger than the reference value, then it was used by the Condenser thread. However, if it was an earlier time value than the reference, then it was ignored by the Condenser thread. Unfortunately, this method required a large number of reads and processing on packets that were not even being used by the Condenser component.

Noting this inefficiency, the new design relies upon the use of file-offset pointers to help trace processed and unprocessed packet records. These pointers use a file-offset value in order to traverse the storage file. For example, a file-offset value of '0' moves the pointer to the beginning of the file. As the Storage Handler's read method is called, the pointer progresses through the file until it reaches the end of the file. Alternatively, the Storage Handler's writing process uses a separate pointer value, but that pointer is typically set to simply track the end of the file. Setting the write file-offset pointer to track the end of the file the Storage Handler is able to ensure that newer entries will occur at the bottom of the storage file.

When the Storage Handler's initialization method opens each file, the read file-offset pointer and the write file-offset pointer values are set to the end of the file. This guarantees that all writes happen at the bottom of the file, and that all reads retrieve records that correspond to the current condensing session. As described above, the write file-offset pointer will continue to follow the end of the storage file as more packet records are stored. However, unlike the write file-offset pointer, the read file-offset pointer only moves during a call from the Condensing thread.

The Condenser process calls the Storage Handler's read method in order to retrieve the packet records between the read file-offset pointer and the write file-offset pointer. Once the records have been returned, the read file-offset pointer is updated with the value that corresponds with the end of the file, and the write file-offset pointer value is allowed to continue being updated as the Dissector threads add new records to the storage file, until the Storage Handler's read method is started again.

3.1.5 Condenser Component

The Condenser component is the final step in the condensing process, before the results are forwarded on to the AMPLIFIER. It is here, that the main purpose for the

CONDENSER is realized, and the lump of received data is reduced down to the Condenser-Report, which consists of a payload and a list of compliant sources.

This abridgment of data happens in a few different steps. First, the data is retrieved from the storage files. In the previous section, the Storage Handler component's retrieval procedure was described. However, one additional program element needs to be explained, so that the advantages of the overall Storage Handler and Condenser components' design can be better understood.

The original design called for multiple threads to access the storage files and then to condense all of the information at once. Unfortunately, this would have forced the Dissector threads to suspend their activities while waiting for access to the storage files. Obviously, this would have led to the Dissector threads being unable to fetch newly arrived packets and process them. Consequently, this could have resulted in a potentially dangerous bottleneck that would have led to many packets being dropped by the Capture component. Additionally, the extra precautions and measures that would have been needed to ensure that the information, retrieved by the Condenser component from each of the storage files, maintained its integrity would increase the amount of complexity inherent in the program.

While the single-threaded Condenser component may be slower in its processing, it offers a solution to the dilemmas faced by the original design. The use of a single thread guarantees that only one storage file can be accessed at a time. This assurance allows the other storage files to remain capable of receiving incoming packets, and thereby removes the bottleneck present in the multi-threaded design. A single-threaded design also means that there is no need to synchronize data received from the storage files, since no interweaving can take place. As a result, the program does not need to include the safe guards that would have been required in the abandoned design, and is consequently able to maintain a lower-level of complexity.

Once the data from the storage files have been retrieved, the second step of the condensing algorithm takes place. In this step, each of the individual retrieved payloads and their characteristics (IP address, port number, flags/UDPPAC keyword) are added to a link list. As each node in the linked list is created, it is assigned an identification number. These ID numbers are given to the "packet digests" to allow for quick reference and finding.

Following the creation of this linked list, the Condenser thread begins parsing the payloads of each of the nodes. The parsing uses a small set of tokens, "0A", "0D", and "20", in order to break the entire payload into more manageable chunks. Remembering that the payload has been converted to a string of hex-represented data, these three tokens represent the following: "0A" = New Line Feed, "0D" = Carriage return, and "20" = Space. These values were chosen as tokens because most plaintext English and other types of communication naturally contain these byte sequences. For that reason, it is very likely that a large payload can be cut up into smaller pieces, and thereby make it easier to locate similarities among the payloads.

After all of the payloads have been parsed into as many small pieces as possible, the Condenser begins to organize the data chunks into groups. This group identification process is facilitated through the use of a trie data-structure. A trie is a data-structure that is often used by dictionaries, spell-checkers, and other similar types of programs. The data-structure itself is very similar to a binary tree data-structure, except that entries can share nodes with other entries and a node can have more than two children. These properties allow for a quicker lookup time than that of the binary search tree. In fact, "looking up a key of length m takes worst case O(m) time. A binary search tree takes O(log n) time, where n is the number of elements in the tree, because lookups depend on the depth of the tree, which is logarithmic in the number of keys[11]."

While Figure 5 does not contain enough entries to demonstrate, Tries also take up less space when they are used to store a number of short strings. This is because nodes are reused

between common initial subsequences [11]. This is particularly true in this case because the data chunks being stored in this trie are bound to have a number of common bytes, resulting from the hex-conversion process substantially reducing the number of possible sequences. Figure 5 demonstrates how a couple of different byte sequences (414E, 4954, and 4953) would be stored in a trie versus a Binary Search Tree.

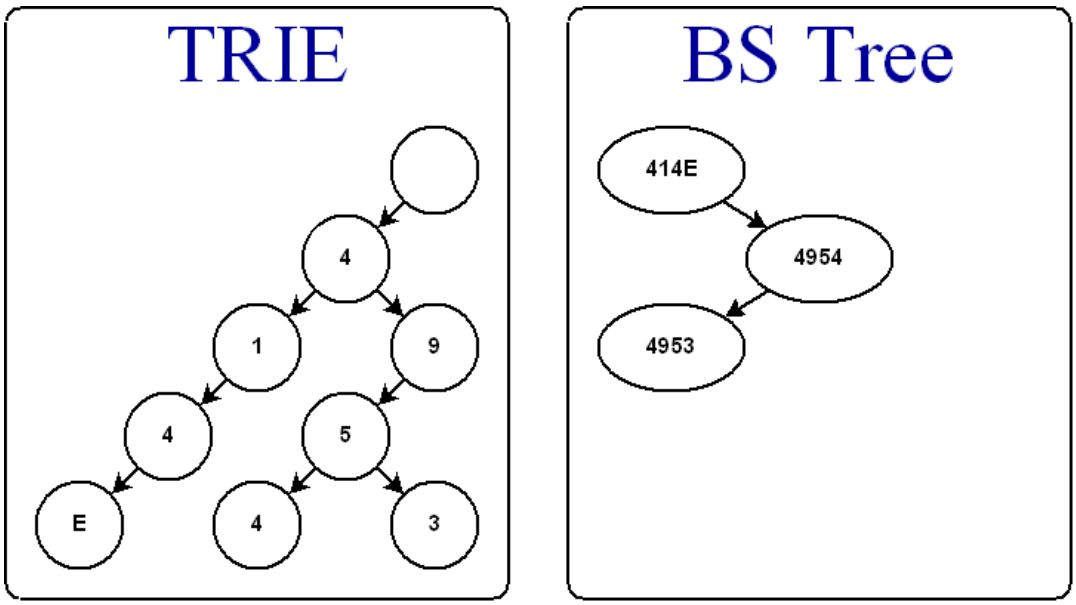

Figure 5. Byte Sequences (414E, 4954, and 4953) Being Stored in a Trie and a Binary Search Tree

At the conclusion of the trie formation, the Condenser component totals the number of data chunks that were stored in the trie and what packets contained each data chunk. For each data chunk that is located in the trie, the packets that contained the chunk receive a certain number of points. The point values are calculated using this small set of rules:

1. The number of packets containing the data chunk must reach a certain user-specified threshold, or a score of zero will added to each packet's score. The threshold value is calculated by dividing the number of packets that contain the data chunk, by the total number of packets. The threshold value can be set by

using the "-o" option when starting the CONDENSER, or the default value of "0.20" will be applied.

2. Once the threshold has been passed, then all packets that contain the data chunk receive the same number of points as there are packets that contain the data chunk. This means that if the threshold was met, and ten packets contained a certain data chunk, then each of those ten packets will receive ten points to be added to their total score.
3. If the data chunk does not have a size of at least 2 bytes, then it is worth zero points.

After the individual packets have been given all of their points and their total scores have been tabulated, the Condenser thread searches for the packet with the highest score. The packet with the maximum score is compared to the CONDENSER's bottom threshold. If the maximum score is not more than this threshold, then the packets' payloads are too random, and the results can not be trusted. The Condenser thread will issue an ERROR message and will discard the condensing algorithms results. If this low-watermark threshold is met, than a third threshold value is used to identify those packets that contained similar payloads. It is also important to note that this low-watermark threshold is a user-modifiable value and it can be set using the "–b" option when starting the CONDENSER, otherwise the default value of "0.05" will be used.

The similarity identification is done by comparing the scores of highest-scoring payload with each of the other packets. If the difference between the maximum score and the score of a packet are within the range allowed by the top threshold, then the IP address and port number of the packet are added to a list. The user can set this top threshold value, also, by using the "–t" option when starting the CONDENSER, or the default value of "0.05" will be used.

Once the list is completed, then it and the payload of the packet with the maximum score are sent to the Condenser/Amplifier Communication component. Aside from the list of IP addresses, port numbers, and the payload, the keyword string, “CONDENSE” is also sent. The IP addresses and port numbers are separated from the payload through the use of the two keyword strings, “<PAYLOAD>” and “</PAYLOAD>”, which are respectively placed at the beginning and end of the payload. Once the Condenser/Amplifier Communication component has been sent the Condense packet, then the Trie and linked list are reset and made ready for the next round of Condensing.

This condensing algorithm was created after researching other approaches used in similar situations. It was during this exploration that some differences between the requirements of the CONDENSER and other pattern-recognition applications became apparent. Although other pattern-recognition programs such as spell-checkers, text-recognition software, and other programs produce reliable results, most of them also require a pre-loaded dataset with which to compare the inputted data. For example, it is extremely problematic for a program to recognize the miss-spelling of any word, if it does not have a dictionary with which to compare. Likewise, DNA sequence recognition software relies upon a preloaded dataset [12]. Many of these other algorithms use complex logic to reach the extreme tolerances needed to fulfill their requirements. However, these algorithms are not directly applicable to the task of condensing or obfuscate the clarity required by a process that may need to be maintained or adapted by future developers.

Therefore, the previously described condensing algorithm was chosen because of its ability to classify inputted data without the use of dictionary, its simple design, and its ability to work in a multi-threaded environment. Yet, the chosen algorithm also represents some of the applicable technology used by other software applications, namely the Trie data-structure and search algorithms [13]. Consequently, the accepted design implements many new ideas, but also incorporates proven methods, in order to ensure quick and accurate results.

3.1.6 Condenser/Amplifier Communication Component

The Condenser/Amplifier Communication component serves as the crucial link between the AMPLIFIER and the CONDENSER. It is through this link that the results from the other CONDENSER components can be transmitted to the AMPLIFIER. It is also through this link that the AMPLIFIER is able to send important messages to the CONDENSER.

When starting the CONDENSER, it is important that the user specify which IP address is to be used for hosting the Condenser/Amplifier Communication server. This specification can be made at startup using the "–a" option or the user can wait to be prompted by the program. This specification requirement allows for the server to be hosted on a NIC other than the one responsible for capturing all of the incoming packets from the targeted hosts, thereby lessening the traffic load. The link is established using a TCP socket connection through the default port of 7777, although the port number can also be user specified by utilizing the "–p" command line option. Once the connection is made, the CONDENSER and AMPLIFIER can maintain a constant route of communication supported by the TCP suite. This ensures that important information will be delivered both timely and accurately to both the CONDENSER and AMPLIFIER.

Previous descriptions have discussed some of the information that the CONDENSER sends to the AMPLIFIER, but the CONDENSER also receives important information from AMPLIFIER. For example, the CONDENSER receives setup information from the AMPLIFIER describing the number of packets that will be initially sent to the intended targets, and the amount of time that should transpire before condensing the received packets. When sending this setup information, the AMPLIFIER precedes it with the keyword, "SETUP." Once, the CONDENSER has received all of the initialization information and has started up all of its components, the keyword, "READY," is sent back to the AMPLIFIER.

This guarantees that the CONDENSER is in a state which is ready to receive the replies of all of the intended targets, before the AMPLIFIER launches the attack packets.

In addition to receiving commands that help pass important setup information, the Condenser/Amplifier Communication component can also receive a termination signal. The keyword, "SHUTDOWN", will cause the Condenser/Amplifier communication thread to notify the Master component and thereby initiate the shutdown sequence. The Master component ensures that the Condenser/Amplifier Communication thread is one of the last threads to close, so that any pertinent data can be transmitted to the AMPLIFIER.

Since the amount of data that will be received by the CONDENSER from the AMPLIFIER is fairly small, the receiving buffer is fairly small compared to the sending buffer. The Condenser/Amplifier Communication component's receiving buffer has a set size of 64 bytes. This is a sufficient size to capture the needed instructions from the AMPLIFIER, and still maintain a limited memory profile.

When data is transmitted from the CONDENSER to the AMPLIFIER prior to shutdown or during routing transmissions, it is first stored in sending buffer. Since the buffer can be written to by multiple components, it is important that access to it is properly controlled. Using the same synchronization techniques that have been described in previous sections, the Condenser/Amplifier Communication component manages both read and write access to the buffer. Another important design aspect of the buffer is that it is able to increase its size if a needed message is too large to be placed into the buffer. This is an especially significant feature if a Condensed payload is both large and common among several different packets. Under a previous design, the oversized Condenser-Report would never have been transmitted and ultimately would have led to a deadlock situation, thus stopping all CONDENSER-to-AMPLIFIER transmissions.

Another important aspect of the current design is its ability to help minimize the amount of system memory needed to meet the requirements of the buffer. This "saved"

memory can be used by the system to increase the amount of resources available to the other CONDENSER components. This memory-saving feature would not have been realized if the default size of the buffer was made extremely large at startup, as suggested by another strategy. However, it is also important to point out that the dynamic size of the buffer is kept from exceeding 16 MB. This upper bound helps to lessen the risk of too much system memory being allocated to the Condenser/Amplifier Communication component's sending buffer, therefore mitigating the potential for segmentation faults and memory exhaustion.

3.1.7 Command/Control Communication Component

The Command/Control Communication component is an added communication feature to the CONDENSER. This component allows for instructions, outside of those issued by the AMPLIFIER, to be sent to the CONDENSER. The component also allows for the CONDENSER to send ERROR, DEBUG, and INFO type messages to a remote client.

While the connection to a remotely controlling host is not required for the CONDENSER to work, it is a useful capability that enables for even greater customization of the CONDENSER program. For example, the remote host is able to adjust the threshold values used by the Condenser component after the program has begun. This is extremely helpful because researchers are able to adjust the amount of data returned in the Condenser-Report, without having to restart the CONDENSER. Researchers are also capable of receiving important messages that are outputted by the CONDENSER concerning certain changes in program status. Using the message type, researchers can identify alerts and other valuable cues by doing such things as color coding the text of the messages displayed by the remote client, or by using other audio or visual indicators. This will help identify any possible errors or anomalies that may be happening with the CONDENSER.

Aside from the few advantages that have been listed here, the Command/Control Communication component is designed to ease the integration of the CONDENSER into

GUI control clients such as Deep Freeze [14]. It would be particularly advantageous if the future control client were to include capabilities for simultaneous communication with both the CONDENSER and AMPLIFIER. While the Command/Control Communication server has been built into the CONDENSER, the client is left for future development.

CHAPTER 4. CONCLUSION

In order for AVALANCHE to boost the research capabilities of ISEAGE, there were a number of aspects that needed to be addressed. Unfortunately, the AMPLIFIER and the Attack Plane portions of AVALANCHE are still in the process of being completed (at the time of this writing), therefore the CONDENSER's capabilities had to be verified using third-party packet-injection software.

Nemesis is a program that allows the user to hand-craft and send packets to user-specified targets. It is also capable of being scripted and therefore can be used to generate thousands of packets very quickly [15]. The CONDENSER verification testing utilized each of these capabilities to verify that the CONDENSER was able to gather targeted-host responses, reduce the reply-payloads into a single packet, calculate received-packet flow statistics, create a persistent record of received-payloads and their identifying characteristics, maintain on-the-fly monitoring capability through status messages. In short, the design and implementation of the CONDENSER address each of the set requirements and meet the quality standards set by the ISEAGE project.

With the completion of the AMPLIFIER and the Attack Plane, AVALANCHE will be able to utilize the full potential of the CONDENSER. In particular, it will be able to simulate many of the distributed and flooding-based attacks. This capability will allow researchers the capacity to better understand these threats and to develop improved solutions to these problems.

While the CONDENSER is currently suited to meet the many requirements that have been outlined, there are a few additional features that could enhance its capabilities even further. The following section will discuss some of these features and how they will work to improve the abilities of scientists and professionals.

4.1 Future Work

Although the CONDENSER has been designed to communicate with a remote Command/Control client, no advanced client has been developed. DeepFreeze was developed in order to serve as a command and control interface for ISEAGE, but it has not yet been integrated into AVALANCHE [14]. It is anticipated that the completion of the other AVALANCHE components will allow for the creation of a single application interface for the whole of AVALANCHE. This approach will allow all status messages and control options for all of AVALANCHE to be located within a single GUI. Until then, the CONDENSER's Command/Control Communication Server can be accessed using a simple telnet client. The data structures that can be used to manipulate the CONDENSER options available to the Command/Control Communication Client are diagramed in Appendix C.

The ability to manipulate the returned Condenser-Report contents is a feature that could be realized through modification to the Command/Control Communication component and the Condenser component. It is probable that the most common packet received will not be the one that researchers want to return. For instance, if the majority of attacks return "failure" payloads, it would be useful to filter them out. This could be accomplished by removing payloads that contain certain keyword data chunks, or even choosing to only return payloads that include a certain keyword data chunk. This will give the researchers a major method of affecting the outcome and contents of the Condenser-Report.

In addition to the creation of an advanced Command/Control client and changes to the Condenser component, the CONDENSER could benefit from improved TCP stream handling. In the current implementation, TCP packets that have been fragmented are treated as individual packets. Although fragmented packets are extremely unlikely if not impossible in ISEAGE, future work should identify a method that will allow each of the fragmented packets to be reassembled into the original packet. In keeping with the overall design goals and architecture, the reassembly method would best be addressed using an additional

Reassembly thread that can coordinate with the Dissector threads. Another possibility for addressing the problem could involve the Attack Plane. Using the concept of "store and forward" routing, it would make sense to employ this capability at the Attack Plane level. Regardless of the manner of its implementation, the realization of this concept is one that would help to increase the realism present in the attack simulation provided by AVALANCHE.

Lastly, the future implementations of additional network protocols would enhance the power of the CONDENSER and AVALANCHE. Since TCP and UDP, are the most likely forms of traffic to be used in the attack simulations, they are the only protocols currently understood by the CONDENSER or AVALANCHE. Future modifications to the Dissector and Capture threads will allow for the additions of new protocols, and thereby increase the realm of study for researchers.

While each of these features and modifications will help make the CONDENSER even more helpful in the cyber-arms race that currently rages between attackers and defenders, the CONDENSER's current capabilities and design will give researchers a tool through which they can study many of the most overwhelming assault methods and tactics facing information assurance professionals, today.

APPENDIX A. BSD PACKET FILTER HELPS

This document, [16], has been included in the appendix to aid the user in creating their BPF statements, if needed. The document was originally written to aid in the creation of filter statements for the TCPDUMP software; however, the filter construction principles are identical to those used by the CONDENSER.

TCPDUMP filters

expression
selects which packets will be dumped. If no *expression* is given, all packets on the net will be dumped. Otherwise, only packets for which *expres sion* is `true' will be dumped.

ber) preceded by one or more qualifiers. There are three different kinds of qualifier:

type qualifiers say what kind of thing the id name or number refers to. Possible types are **host**, **net** and **port**. E.g., `host foo', `net 128.3', `port 20'. If there is no type qualifier, **host** is assumed.

dir qualifiers specify a particular transfer direction to and/or from *id*. Possible directions are **src**, **dst**, **src or dst** and **src and dst**. E.g., `src foo', `dst net 128.3', `src or dst port ftp-data'. If there is no dir qualifier, **src or dst** is assumed. For `null' link layers (i.e. point to point pro tocols such as slip) the **inbound** and **out bound** qualifiers can be used to specify a desired direction.

proto qualifiers restrict the match to a particu lar protocol. Possible protos are: **ether**, **fddi**, **tr**, **ip**, **ip6**, **arp**, **rarp**, **decnet**, **tcp** and **udp**. E.g., `ether src foo', `arp net 128.3', `tcp port 21'. If there is no proto qualifier, all protocols consistent with the type are assumed. E.g., `src foo' means `(ip or arp or rarp) src foo' (except the latter is not legal syntax), `net bar' means `(ip or arp or rarp) net bar' and `port 53' means `(tcp or udp) port 53'.

[`fddi' is actually an alias for `ether'; the parser treats them identically as meaning ``the data link level used on the specified network interface." FDDI headers contain Ethernet-like source and destination addresses, and often contain Ethernet-like packet types, so you can filter on these FDDI fields just as with the analogous Ether net fields. FDDI headers also contain other fields, but you cannot name them explicitly in a filter expression.

Similarly, `tr' is an alias for `ether'; the previ ous paragraph's statements about FDDI headers also apply to Token Ring headers.]

In addition to the above, there are some special `primitive' keywords that don't follow the pattern: **gateway**, **broadcast**, **less**, **greater** and arithmetic expressions. All of these are described below.

tives. E.g., `host foo and not port ftp and not port ftp-data'. To save typing, identical quali fier lists can be omitted. E.g., `tcp dst port ftp or ftp-data or domain' is exactly the same as `tcp dst port ftp or tcp dst port ftp-data or tcp dst port domain'.

Allowable primitives are:

dst host *host*
True if the IPv4/v6 destination field of the packet is *host*, which may be either an address or a name.

src host *host*
True if the IPv4/v6 source field of the packet is *host*.

host *host*
True if either the IPv4/v6 source or desti nation of the packet is *host*. Any of the above host expressions can be prepended with the keywords, **ip**, **arp**, **rarp**, or **ip6** as in:
ip host *host*
which is equivalent to:
ether proto *ip* **and host** *host*
If *host* is a name with multiple IP addresses, each address will be checked for a match.

ether dst *ehost*
True if the ethernet destination address is *ehost*. *Ehost* may be either a name from /etc/ethers or a number (see **ethers(3N)** for numeric format).

ether src *ehost*
True if the ethernet source address is *ehost*.

ether host *ehost*
True if either the ethernet source or desti nation address is *ehost*.

gateway *host*
True if the packet used *host* as a gateway. I.e., the ethernet source or destination address was *host* but neither the IP source nor the IP destination was *host*. *Host* must be a name and must be found both by the machine's host-name-to-IP-address resolution mechanisms (host name file, DNS, NIS, etc.) etc.). (An equivalent expression is
ether host *ehost* **and not host** *host*
which can be used with either names or num bers for *host / ehost*.) This syntax does not work in IPv6-enabled configuration at this moment.

dst net *net*
True if the IPv4/v6 destination address of the packet has a network number of *net*. *Net* may be either a name from /etc/networks or a network number (see *networks(4)* for details).

src net *net*
True if the IPv4/v6 source address of the

Figure 6. BPF Help Part 1 [16]

packet has a network number of *net*.

net *net*
True if either the IPv4/v6 source or destination address of the packet has a network number of *net*.

net *net* **mask** *netmask*
True if the IP address matches *net* with the specific *netmask*. May be qualified with **src** or **dst**. Note that this syntax is not valid for IPv6 *net*.

net *net*/*len*
True if the IPv4/v6 address matches *net* with a netmask *len* bits wide. May be qualified with **src** or **dst**.

dst port *port*
True if the packet is ip/tcp, ip/udp, ip6/tcp or ip6/udp and has a destination port value of *port*. The *port* can be a number or a name used in /etc/services (see **tcp(4P)** and **udp(4P)**). If a name is used, both the port number and protocol are checked. If a number or ambiguous name is used, only the port number is checked (e.g., **dst port 513** will print both tcp/login traffic and udp/who traffic, and **port domain** will print both tcp/domain and udp/domain traffic).

src port *port*
True if the packet has a source port value of *port*.

True if either the source or destination port of the packet is *port*. Any of the above port expressions can be prepended with the keywords, **tcp** or **udp**, as in:
tcp src port *port*
which matches only tcp packets whose source port is *port*.

less *length*
True if the packet has a length less than or equal to *length*. This is equivalent to:
len <= *length*.

greater *length*
True if the packet has a length greater than or equal to *length*. This is equivalent to:
len >= *length*.

ip proto *protocol*
True if the packet is an IP packet (see **ip(4P)**) of protocol type *protocol*. *Protocol* can be a number or one of the names *icmp*, *icmp6*, *igmp*, *igrp*, *pim*, *ah*, *esp*, *vrrp*, *udp*, or *tcp*. Note that the identifiers *tcp*, *udp*, and *icmp* are also keywords and must be escaped via backslash (\), which is \\ in the C-shell. Note that this primitive does not chase the protocol header chain.

ip6 proto *protocol*
True if the packet is an IPv6 packet of protocol type *protocol*. Note that this primitive does not chase the protocol header chain.

ip6 protochain *protocol*
True if the packet is IPv6 packet, and contains protocol header with type *protocol* in its protocol header chain. For example,
ip6 protochain 6
matches any IPv6 packet with TCP protocol header in the protocol header chain. The packet may contain, for example, authentication header, routing header, or hop-by-hop option header, between IPv6 header and TCP header. The BPF code emitted by this primitive is complex and cannot be optimized by BPF optimizer code in *tcpdump*, so this can be somewhat slow.

ip protochain *protocol*
Equivalent to **ip6 protochain** *protocol*, but True if the packet is an ethernet broadcast packet. The *ether* keyword is optional.

ip broadcast
True if the packet is an IP broadcast packet. It checks for both the all-zeroes and all-ones broadcast conventions, and looks up the local subnet mask.

ether multicast
True if the packet is an ethernet multicast packet. The *ether* keyword is optional. This is shorthand for `**ether[0] & 1 != 0**'.

ip multicast
True if the packet is an IP multicast packet.

ip6 multicast
True if the packet is an IPv6 multicast packet.

ether proto *protocol*
True if the packet is of ether type *protocol*. *Protocol* can be a number or one of the names *ip*, *ip6*, *arp*, *rarp*, *atalk*, *aarp*, *decnet*, *sca*, *lat*, *mopdl*, *moprc*, *iso*, *stp*, *ipx*, or *netbeui*. Note these identifiers are also keywords and must be escaped via backslash (\).

[In the case of FDDI (e.g., `**fddi protocol arp**') and Token Ring (e.g., `**tr protocol arp**'), for most of those protocols, the protocol identification comes from the 802.2 Logical Link Control (LLC) header, which is usually layered on top of the FDDI or Token Ring header.

When filtering for most protocol identifiers on FDDI or Token Ring, *tcpdump* checks only the protocol ID field of an LLC header in

Figure 7. BPF Help Part 2 [16]

so-called SNAP format with an Organizational Unit Identifier (OUI) of 0x000000, for encapsulated Ethernet; it doesn't check whether the packet is in SNAP format with an OUI of 0x000000.

The exceptions are *iso*, for which it checks the DSAP (Destination Service Access Point) and SSAP (Source Service Access Point) fields of the LLC header, *stp* and *netbeui*, packet with an OUI of 0x080007 and the Appletalk etype.

In the case of Ethernet, *tcpdump* checks the Ethernet type field for most of those proto cols; the exceptions are *iso*, *sap*, and *net beui*, for which it checks for an 802.3 frame and then checks the LLC header as it does for FDDI and Token Ring, *atalk*, where it checks both for the Appletalk etype in an Ethernet frame and for a SNAP-format packet as it does for FDDI and Token Ring, *aarp*, where it checks for the Appletalk ARP etype in either an Ethernet frame or an 802.2 SNAP frame with an OUI of 0x000000, and *ipx*, where it checks for the IPX etype in an Eth ernet frame, the IPX DSAP in the LLC header, the 802.3 with no LLC header encapsulation of IPX, and the IPX etype in a SNAP frame.]

decnet src *host*
True if the DECNET source address is *host*, which may be an address of the form ``10.123'', or a DECNET host name. [DECNET host name support is only available on Ultrix systems that are configured to run DECNET.]

decnet dst *host*
True if the DECNET destination address is *host*.

decnet host *host*
True if either the DECNET source or destina tion address is *host*.

ip, **ip6**, **arp**, **rarp**, **atalk**, **aarp**, **decnet**, **iso**, **stp**, **ipx**, *netbeui*
Abbreviations for:
ether proto *p*
where *p* is one of the above protocols.

lat, **moprc**, **mopdl**
Abbreviations for:
ether proto *p*
where *p* is one of the above protocols. Note that *tcpdump* does not currently know how to parse these protocols.

vlan *[vlan_id]*
True if the packet is an IEEE 802.1Q VLAN packet. If *[vlan_id]* is specified, only encountered in *expression* changes the decod ing offsets for the remainder of *expression* on the assumption that the packet is a VLAN packet.

tcp, **udp**, **icmp**
Abbreviations for:
ip proto *p* **or ip6 proto** *p*
where *p* is one of the above protocols.

iso proto *protocol*
True if the packet is an OSI packet of pro tocol type *protocol*. *Protocol* can be a num ber or one of the names *clnp*, *esis*, or *isis*.

clnp, **esis**, **isis**
Abbreviations for:
iso proto *p*
where *p* is one of the above protocols. Note that *tcpdump* does an incomplete job of pars ing these protocols.

expr relop expr
True if the relation holds, where *relop* is one of >, <, >=, <=, =, !=, and *expr* is an arithmetic expression composed of integer constants (expressed in standard C syntax), the normal binary operators [+, -, *, /, &, |], a length operator, and special packet data accessors. To access data inside the packet, use the following syntax:
proto **[** *expr* **:** *size* **]**
Proto is one of **ether, fddi, tr, ip, arp, rarp, tcp, udp, icmp** or **ip6**, and indicates the protocol layer for the index operation. Note that *tcp, udp* and other upper-layer protocol types only apply to IPv4, not IPv6 (this will be fixed in the future). The byte offset, relative to the indicated pro tocol layer, is given by *expr*. *Size* is optional and indicates the number of bytes in the field of interest; it can be either one, two, or four, and defaults to one. The length operator, indicated by the keyword **len**, gives the length of the packet.

For example, `**ether[0] & 1 != 0**' catches all multicast traffic. The expression `**ip[0] & 0xf != 5**' catches all IP packets with options. The expression `**ip[6:2] & 0x1fff = 0**' catches only unfragmented datagrams and frag zero of fragmented datagrams. This always means the first byte of the TCP *header*, and never means the first byte of an intervening fragment.

Some offsets and field values may be expressed as names rather than as numeric values. The following protocol header field offsets are available: **icmptype** (ICMP type field), **icmpcode** (ICMP code field), and **tcpflags** (TCP flags field).

The following ICMP type field values are available: **icmp-echoreply**, **icmp-unreach**, **icmp-sourcequench**, **icmp-redirect**, **icmp-echo**, **icmp-routeradvert**, **icmp-routersolicit**, **icmp-timxceed**, **icmp-paramprob**, **icmp-tstamp**, **icmp-**

Figure 8. BPF Help Part 3 [16]

tstampreply, **icmp-ireq**, **icmp-ireqreply**, **icmp-maskreq**, **icmp-maskreply**.

The following TCP flags field values are available: **tcp-fin**, **tcp-syn**, **tcp-rst**, **tcp-push**, **tcp-push**, **tcp-ack**, **tcp-urg**.

Primitives may be combined using:

A parenthesized group of primitives and operators (parentheses are special to the Shell and must be escaped).

Negation (`**!**' or `**not**').

Concatenation (`**&&**' or `**and**').

Alternation (`**||**' or `**or**').

Negation has highest precedence. Alternation and concatenation have equal precedence and associate left to right. Note that explicit **and** tokens, not juxtaposition, are now required for concatenation.

If an identifier is given without a keyword, the most recent keyword is assumed. For example,
not host vs and ace
is short for
not host vs and host ace
which should not be confused with
not (host vs or ace)

Expression arguments can be passed to *tcpdump* as either a single argument or as multiple arguments, whichever is more convenient. Generally, if the expression contains Shell metacharacters, it is before being parsed.

EXAMPLES

To print all packets arriving at or departing from *sun down*:
tcpdump host sundown

To print traffic between *helios* and either *hot* or *ace*:
tcpdump host helios and \(hot or ace \)

To print all IP packets between *ace* and any host except *helios*:
tcpdump ip host ace and not helios

To print all traffic between local hosts and hosts at Berkeley:
tcpdump net ucb-ether

To print all ftp traffic through internet gateway *snup*: (note that the expression is quoted to prevent the shell from (mis-)interpreting the parentheses):
tcpdump 'gateway snup and (port ftp or ftp-data)'

To print traffic neither sourced from nor destined for local hosts (if you gateway to one other net, this stuff should never make it onto your local net).
tcpdump ip and not net *localnet*

To print the start and end packets (the SYN and FIN packets) of each TCP conversation that involves a non-local host.
tcpdump 'tcp[tcpflags] & (tcp-syn|tcp-fin) != 0 and not src and dst net *localnet*'

To print IP packets longer than 576 bytes sent through gateway *snup*:
tcpdump 'gateway snup and ip[2:2] > 576'

To print IP broadcast or multicast packets that were *not* sent via ethernet broadcast or multicast:
tcpdump 'ether[0] & 1 = 0 and ip[16] >= 224'

To print all ICMP packets that are not echo requests/replies (i.e., not ping packets):
tcpdump 'icmp[icmptype] != icmp-echo and icmp[icmptype] != icmp-echoreply'

Figure 9. BPF Help Part 4 [16]

APPENDIX B. CONDENSER PACKET STRUCTURES

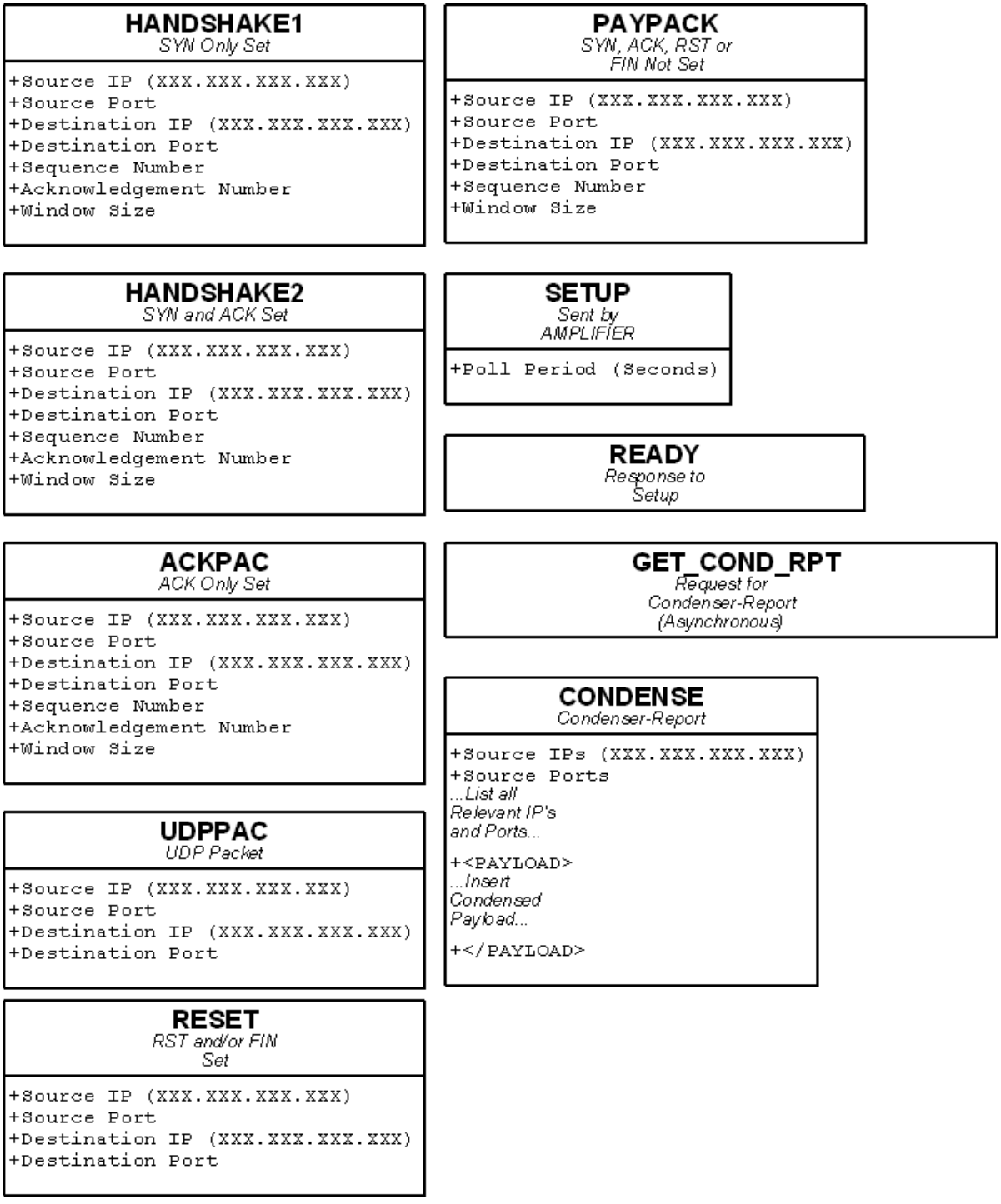

Figure 10. Packet Structures Used for Communication between the AMPLIFIER and the CONDENSER

APPENDIX C. CMD/CNTRL COM PACKET STRUCTURES

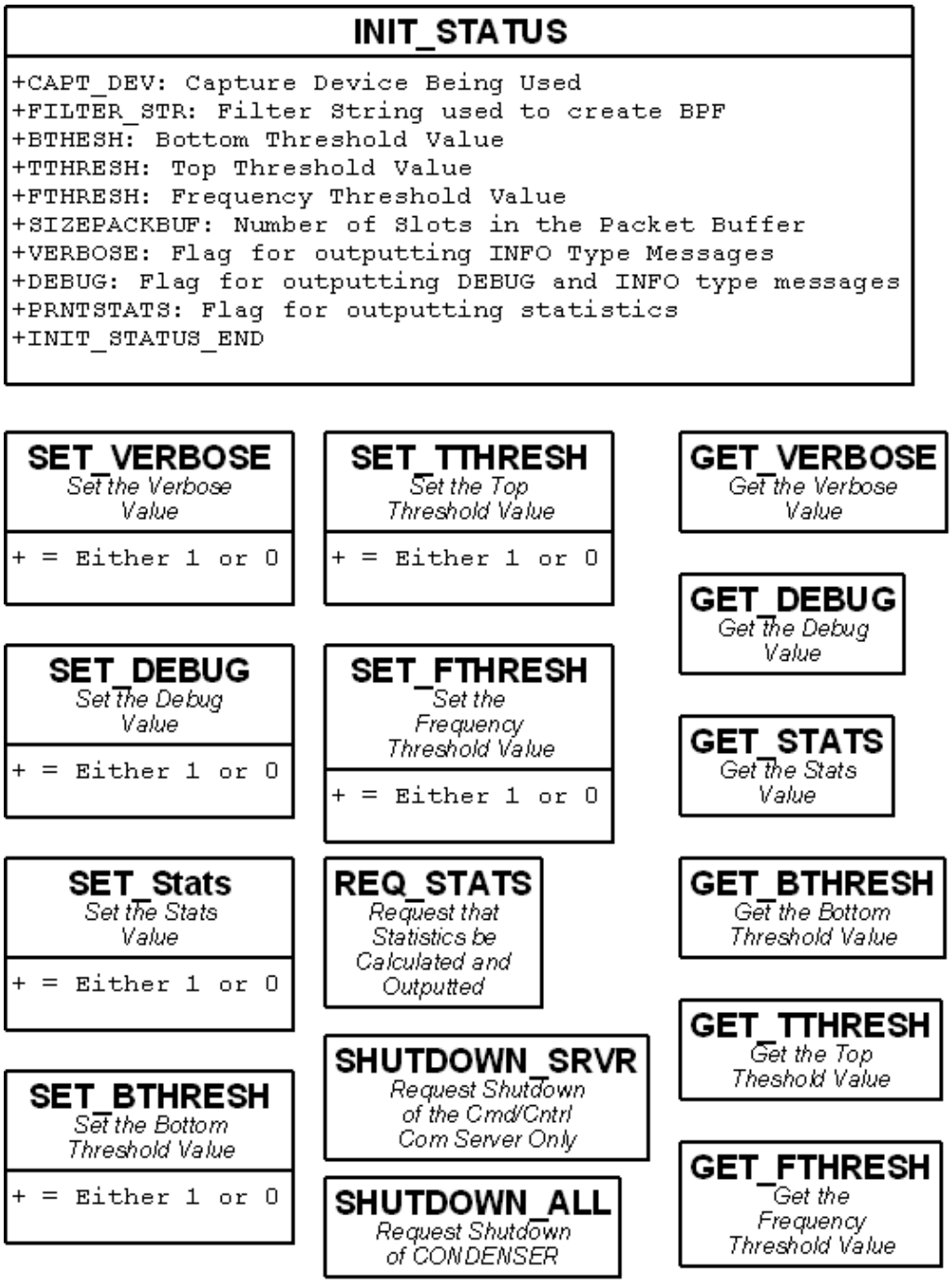

Figure 11. Packet Structures Used for Communication between the Command/Control Communication Server and Client

APPENDIX D. COMMAND-LINE OPTIONS

-V Turn on Verbosity. Output all INFO Statements.

-D Turn on Debug. Output all DEBUG Statements.

-S Turn on Statistics. Output the Packet Statistics.

-A Turn on Append Flag. This keeps the packet storage files from being truncated.

-H Print this Help Section to STDOUT.

-i device Manually set the network card to be used for capturing.
ex: -i eth0

-a "*IP Address*" Manually set the device that will be used for the Cond/Amp Communication Server.
ex: -a "192.168.1.3"

-x *number* Manually set the number of packet buffers that will be used.
default: 2048
ex: -x 24

-p *number* Manually set the port number that the Cond/Amp Communication Server will listen on.
default: 7777
ex: -p 3333

-c "*IP Address*" Manually set the device that will be used for the Cmd/Cntrl Communication Server.
ex: -c "192.168.1.4"

-r *number* Manually set the port number that the Cmd/Cntrl Communciation Server will listen on.
default: 7778
ex: -r 3351

-f "*Filter String*" Manually set the Packet Filter String.
ex: -f "tcp or udp and not src host 192.168.1.100"

-b *number* Manually set the percentage (decimal form) to be used as the bottom threshold of the condenser. (0.0 <= number <= 1.0) are the constraints. The lower the threshold the more variance in the data sample will be allowed.
default: 0.05
ex: -b .03

-**t** *number* Manually set the percentage (decimal form) to be used as the top threshold of the condenser. (0.0 <= number <= 1.0) are the constraints. The lower the threshold the higher in similarity all returned packets must be.
default: 0.05
ex: -t 0.07

-**o** *number* Manually set the percentage (decimal form) to be used as the threshold value for the frequency with which a word must appear in order to be counted. (0.0 <= number <= 1.0) are the constraints. The lower the threshold the fewer the number of packets that must contain the word for it to count.
default: 0.20
ex: -o 0.35

BIBLIOGRAPHY

[1] "Botnet," *Wikipedia,* Mar. 21, 2008. [Online]. Available: http://en.wikipedia.org/wiki/Botnet [Accessed Mar. 22, 2008].

[2] C. C. Zou and R. Cunningham, "Honeypot-Aware Advanced Botnet Construction and Maintenance," *Proc. 2006 Int'l Conf. Dependable Systems and Networks*. (DSN 2006).

[3] P. Bächer et al."Know your Enemy: Tracking Botnets," *Honeynet Project and Research Alliance*, Mar. 13, 2005; [Online]. Available: http://www.honeynet.org/papers/bots/ [Accessed Mar. 22, 2008].

[4] R. Vamosi, "Newsmaker: Cyberattack in Estonia—what it really means," *CNET News.com*, May 29, 2007. [Online]. Available: http://www.news.com/ 2008-7349_3-6186751.html [Accessed Mar. 24, 2008].

[5] "Denial-of-service attack," *Wikipedia*, Mar. 19, 2008. [Online]. Available: http://en.wikipedia.org/wiki/Denial_of_service [Accessed Mar. 22, 2008].

[6] S. McClure, J. Scambray, and G. Kurtz, *Hacking Exposed Fifth Edition: Network Security Secrets & Solutions*, McGraw-Hill/Osborne, 2005, pp. 488-507.

[7] D. Jacobson, "ISEAGE: implementation plan," 16 Jun. 2005. [Online]. Available: http://www.iac.iastate.edu/iseage/implementation_plan.pdf [Accessed Mar. 22, 2008].

[8] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming,* Addison-Wesley, 2000, pp. 1, 3, 153-161.

[9] W. R. Stevens, *TCP/IP Illustrated,* Vol. 1. Reading, MA: Addison-Wesley, 1994.

[10] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming: The Sockets Networking API*, 3rd ed., Vol. 1. Boston. Addison-Wesley, 2004, pp. 788-790.

[11] "Trie," *Wikipedia*, Mar. 4, 2008. [Online]. Available: http://en.wikipedia.org/wiki/Trie [Accessed Mar. 22, 2008].

[12] Q. Ma, J.T.L. Wang, D. Shasha, and C. H. Wu, "DNA sequence classification via an expectation maximization algorithm and neural networks: a case study," *IEEE Trans. on Systems, Man, and Cybernetics, Part C: Applications and Reviews,* vol. 31, issue 4, pp. 468-475, Nov. 2001. [Online]. Available: http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=983930 [Accessed Apr. 4, 2008].

[13] J.A. Iglesias, A. Ledezma, A. Sanchis, "Sequence classification using statistical pattern recognition," in *Lecture Notes in Computer Science: Advances in Intelligent Data Analysis VII*, Vol. 4723/2007, Springer Berline/Heidelberg, 2007, pp. 207-218. [Online] Available: http://www.springerlink.com/content/85tn7411658449hp/ [Accessed Apr. 4, 2008].

[14] N. L. Karstens, "DeepFreeze: a management interface for ISEAGE," M.S. thesis, Iowa State University, Ames, IA, United States of America, 2007.

[15] "Nemesis" *Sourceforge.net*. [Online]. Available: http://nemesis.sourceforge.net/index.html [Accessed Mar. 24, 2008].

[16] M. Iliofotou, "Tcpdump filters," [Online]. Available: http://www.cs.ucr.edu/~marios/ethereal-tcpdump.pdf [Accessed Mar. 22, 2008].

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank the members of my committee for not only the many hours and effort spent in regards to this project, but the many other projects, classes, and activities that I have been able to enjoy at Iowa State. In particular, I would like to thank Dr. Doug Jacobson for the extreme effort that he contributes to ensure the success of ISEAGE and Cyber Corps programs. I would also like to thank the many other professors and peers that have taught me many things both in and out of the Information Assurance field.

Most importantly, I want to thank my dear wife and daughter for their unending support and sacrifice on my behalf. I also want to thank my wonderful parents who have given so much in time, effort, and patience, so that I could achieve this goal. Truly this accomplishment is only evidence of the many blessings and people that have helped me in my life.